TX-0 COMPUTER
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS 02139

M-5001-39-1

THE MIDAS ASSEMBLY PROGRAM

August 22, 1966

THE MIDAS ASSEMBLY PROGRAM

THE MIDAS ASSEMBLY PROGRAM

Introduction

Programming for a digital computer is writing the precise sequence of instructions and data which is required to perform a given computation. The purpose of an assembly program is to facilitate programming by translating a source language, which is convenient for the programmer to use, into a numerical representation or object program, which is convenient for the computer hardware to deal with. A symbolic assembly program such as MIDAS permits the programmer to use mnemonic symbols to represent instructions, locations, and other quantities with which he may be working. The use of symbolic labels or address tags permits the programmer to refer to instructions or data without actually knowing or caring what specific location in the computer memory they may occupy.

MIDAS is a two pass assembler; that is, it normally processes the source program twice. During the first pass, it enters all symbols definitions encountered into its symbol table, which it then uses on Pass 2 to generate the complete object program.

The MIDAS Source Language

A program consists of a sequence of numbers in memory which may be instructions, data, or both. We shall refer to these numbers as words without specifying whether they are instructions or not. A word is denoted in the source program by one or more syllables separated by suitable combining operators, and terminated by a tab or carriage return. A syllable may be defined

as being the smallest element of the programming language which has a numerical or operational value. The following are some different types of syllables:

1. Integers. An integer is a string of digits, which will be interpreted as an octal or decimal number.
2. Symbols. A symbol is a string of alphanumeric characters (lower case numerals and letters, and upper case letters except U, I, S, and X) containing at least one letter. The first six characters of a symbol are used to identify it if it is more than six characters long.

Syllables may be combined with the following operators:

+ or space means addition, modulo $2^{18}-1$ (one's complement).

- means addition of the one's complement.

U means logical union (inclusive or).

I means logical intersection (logical and).

S means logical disjunction (exclusive or).

X means integer multiplication.

A symbolic expression is one syllable, or more than one syllable combined with these operators. We shall refer to +, -, and space as additive operators, and U, I, S, and X as product operators.

Operations are performed from left to right, except all product operations are performed before additive operations. It is not admissible to precede or follow a product operator with any other operator. In a string of consecutive additive operators, the last one seen applies.

The following examples of symbolic expressions on the left have the value listed on the right. (All numbers in this memo are octal unless followed by a decimal point ".".)

2	2
2+3	5
2-3	777776
2X3	6
2U3	3
2I3	2
2S3	1
-2S3	777776
--1	777776
- 1	1
7-2U3	4
add 40	200040
calUcom	700240

A symbolic expression terminated by a tab or carriage return is a storage word. The location in memory to which it is assigned is determined by a location counter in MIDAS. After each word is assigned, the location counter is advanced by one.

More About Symbols. Pseudo-Instructions.

MIDAS classifies symbols according to the manner of their definition. The initial vocabulary consists of symbols for the more commonly used TX-0 instructions, and also a class of symbols called pseudo-instructions, which represent directions to MIDAS on how to proceed with the assembly. Some examples of pseudo-instructions are:

P-I	Action
octal	All integers following (unless specifically denoted as decimal) are interpreted as octal numbers until next appearance of pseudo-instruction decimal.
decimal	All integers following are interpreted as decimal numbers until next appearance of pseudo-instruction octal.

start Denotes the end of the program.

Additional pseudo-instructions will be discussed at opportune places. A complete list is given in Appendix 1.

Symbols are defined in the following ways:

1. As address tags. A comma following a symbolic expression denotes an address tag. If the tag is a single, undefined symbol, it will be defined with numerical value equal to the present value of the location counter. If the tag is any other defined symbolic expression, it will have its value compared with the present value of the location counter, and an error comment (mdt) will be made in the event of a disagreement. If the tag is any other symbolic expression which is undefined when encountered on Pass 2, an error comment is made (ust). Use of a defined symbol as an address tag cannot change the value of the symbol.

2. By parameter assignments. A symbol may be assigned a numerical value by the use of a parameter assignment.

   a. The form

      symbol=expr$_{\Delta}$

      where symbol is any legal symbol and expr is any symbolic expression terminated by a tab or carriage return, defines symbol as having the numerical value of expr. Parameter assignments may be used to set table sizes, define new operation codes, or for other purposes. Thus

      cle=calUxro$_{\Delta}$

      defines cle as 700201, which, as an operate instruction, would clear the AC, LR, and XR.

   b. A symbol may be defined as an "initial" symbol, and hence will not be printed by the symbol print, by using: instead of =. Opsyn and equals will preserve this distinction.

3. As variables. The appearance of a letter or letters in upper case in any legal, undefined symbol, at any appearance of that symbol, defines that symbol as a variable. For each such symbol defined, one register is allocated in a region of storage reserved by the next appearance of the pseudo-instruction variables. The initial contents of these registers is undefined. This feature facilitates the reserving of temporary storage locations. Example:

```
        .
        .
        .
sto temp
tsx subr
lda Temp
        .
        .
        .
variables
```

Beware: Do NOT try to use upper case letters U, I, S, or X to denote a variable!

4. As macro instructions. A symbol is defined as a macro-instruction name by use of the pseudo-instruction define. Further discussion of macro instructions will be left until later.

5. With equals or opsyn. A symbol may be defined as precisely equivalent to any other symbol by use of the pseudo-instruction equals and opsyn. The usage is:

```
equals anysym, defsym
        or
opsyn anysym, defsym
```

where the symbol anysym is made logically equivalent to defsym if the latter is defined. Previously defined symbols are redefined. Equals and opsyn differ in one respect: opsyn is effective in Pass 1 only. These may be used to define a logical equivalent for any other defined symbol. Thus abbreviations may be defined for pseudo-instructions if desired. Note that

equals and opsyn are NOT the same as the equals sign used in parameter assignments, and are not in general interchangeable with it. Equals and opsyn are used to give a symbol a logical or operational value, while parameter assignments are used to give a symbol a numerical value. Beware that if you define synonyms for start that either the synonym starts with the letters s, t, a, r, t or the word start appears after the use of the synonym at the end of the source program tape. Although the main processor in MIDAS will recognize synonyms for start, the part of the program which reads tape will not, and must be fooled into stopping the tape reader independently of the rest of the assembly.

The Location Counter

The MIDAS location counter records the assigned location for each word in the object program. It is set to 20 at the beginning of each pass, and counts upward modulo memory size. The location counter may be set to any value by writing

expr|

This sets the location counter to the value of the symbolic expression expr modulo 2^{13}. If expr contains an undefined symbol, on Pass 1 the location becomes indefinite, and the definition of address tags is inhibited until the location again becomes definite by means of a defined location assignment. On Pass 2, an undefined symbol will result in an error message (us-). The undefined symbol is taken as zero, and the location remains definite. The pseudo-instruction variables may not be used when the location is indefinite.

The value of the location counter may be obtained by using the special syllable "." (period). Examples:

```
tze .+2                          com
tra nonzero                      trn .-1
llr foo
```

The first example transfers to location nonzero if the AC contains any number other than zero, but zero in the AC causes the program to skip to the llr instruction. The second example puts the magnitude of the contents of the AC into the AC by transferring back to the complement instruction until the AC becomes positive. The second instruction is read "trn point minus one."

The character ".", when preceded by an integer, denotes that the integer is to be considered as decimal regardless of the effect of the pseudo-instruction octal or decimal. "Point" means location counter only when it appears as a distinct syllable. Thus,

add . means add this instruction to AC

20. means 20 decimal.

The character |, when not preceded by an expression, denotes the beginning of a comment. Characters following it are ignored until the next tab or carriage return.

Constants

Constants required by a program will be reserved automatically by MIDAS when enclosed in parentheses. Thus, if it is required to get the number add 20 into the accumulator, one can write

```
lda (add 20)
```

The word enclosed in parentheses is stored in a block reserved by the next appearance of the pseudo-instruction constants.

Duplicate constants are stored only once. Closing parens will be supplied automatically by MIDAS if the character following is a word terminator (e.g., tab or carriage return). The constant word and surrounding parens are treated as a single syllable whose value is the address of a register containing the constant word. Constants may be used in constants. The following two program fragments are equivalent:

```
add (add (20)-11r-(30          add a
      .                          .
      .                          .
      .                          .
constants                  a,    add b-11r-c
                           b,    20
                           c,    30
```

The pseudo-instruction constants may not be used where the location is indefinite.

Flexo Code Pseudo-Instructions

Three pseudo-instructions are provided to facilitate handling flexowriter characters in programs. These are:

1. character qc, where q is any of the letters l, m, or r, which specifies whether the character c is to be placed in the left (bits 0, 3, 6, 9, 12., 15.), middle (bits 1, 4, 7, 10., 13., 16.) or right (bits 2, 5, 8, 11., 14., 17.) portion of the word. The pseudo-instruction, with its argument, is treated as a single syllable.

2. flexo abc, where a, b, c are any three flexo characters, is equivalent to

   character ra+character mb+character lc

3. text qArbitrary string of characters.q, where the arbitrary string of characters is stored three to a word as in flexo until the first character q is encountered again. Neither appearance of q is considered part of the string. Thus q may be any character not appearing in the string.

The following examples demonstrate their usage.

```
character rf  is equivalent to 11010
character mm   "       "     "  222000
flexo thi      "       "     "  100000+202000+004400=306400
text .this.    "       "     "  {306400}
                                  {001010}
```

Macro Instructions

Often certain character sequences appear several times throughout a program in almost identical form. The following example illustrates such a repeated sequence.

```
lda a
add b
sto c
lda d
add e
sto f
```

The sequence:

```
lda x
add y
sto z
```

is the model upon which the repeated sequence is based. This model can be defined as a macro instruction and given a name. The characters x, y, and z are called dummy arguments, and are identified as such by being listed immediately following the macro name when the macro instruction is defined. Other characters, called arguments, are substituted for the dummy arguments each time the model is used. The appearance of a macro-instruction name in the source program is referred to as a call.

The arguments are listed immediately following the macro name when the macro instruction is called. When a macro instruction is called, MIDAS reads out the characters which form the macro-instruction definition, substitutes the characters of the arguments for the dummy arguments, and inserts the resulting characters into the source program as if typed there originally.

The process of defining a macro is best illustrated with and example:

```
define  write a,b
        tsx wr
        b-.-2
        text |a|
b,      terminate
```

The pseudo-instruction define defines the first legal symbol following it as a macro name. Next follow dummy arguments as required, separated by commas, terminated by a tab or carriage return. Next follows the body of the macro definition. The pseudo-instruction terminate indicates the end of the macro definition. Appearances of dummy arguments are marked, and the character string is stored away. Dummy arguments are delimited by the following characters: plus, minus, space, U, I, S, X, upper case, lower case, tab, carriage return, equals, comma, bar, colon, and upper case 1, 6, and 9. Dummy arguments must be legal symbols; any previous definition of dummy argument symbols is ignored while in the macro definition.

A macro call consists of the macro name, followed if desired by a list of arguments separated with commas, and terminated with a tab or carriage return. The write macro, if called as follows:

```
write This gets printed out., nextag
```

generates the following code:

```
        tsx wr
        nextag-.-2
        text |This gets printed out.|
nextag,
```

which with a suitable text-printing subroutine, might comprise the necessary code for printing "This gets printed out." on the flexowriter. The argument to be printed, using this format, must not contain the characters comma, tab, carriage return or bar. Comma, tab, or carriage return would end the argument while bar would terminate the argument of the text pseudo-instruction. So that comma, tab, and carriage return can be used within arguments, the argument quotation characters upper case ⁶ and ⁹ are provided. They might be used as follows:

```
write ⁶This, of course, has commas.⁹
It also has a carriage return, ⁹ , nextag
```

All characters within a pair of argument quotes are considered to be one argument, and this entire argument, with the quotes removed, will be substituted for the dummy argument in the original definition. MIDAS marks the end of an argument only on seeing comma, tab, or carriage return not enclosed within argument quotes. If quotes appear within quotes, the outermost pair is deleted. If an outer argument quote is immediately preceded by an upper case and immediately followed by a lower case, both case shifts are deleted also. A tab or carriage return immediately following a macro name denotes that no arguments are read. Any other separating character will be the first character of the

first argument except space: a space used as a separator will be deleted and will not be part of the first argument.

The second argument of the write macro is a symbol which is defined as an address tag each time the macro is called, so a different symbol must be supplied at each call of the macro to avoid multiply defined tags. MIDAS will supply suitable created symbols for this purpose, guaranteed to be unique to each call of the macro, if we write the first line of the definition thusly:

define write a|b or define write a,|b

In either case, the vertical bar denotes dummy symbols following it will be supplied from special created symbols if not explicitly supplied when the macro is called. The created symbols are of the form 000a01, 000a02,... 000a09, 000a0a, etc. The created symbol generator is reset to 000a01 at the beginning of each pass. The number of created symbols may not exceed 33,695.. Note that unsupplied arguments corresponding to dummy arguments preceding the bar are plugged in as empty strings. Supplied arguments corresponding to dummy arguments following a bar suppress the generation of a corresponding created symbol.

There remains one problem: How do we plant dummy arguments in the argument of character r, m, or l? Of course, the r, m, or l could be part of the supplied argument, but there is another way. Write, say:

```
define macro a
          .
          .
          .
     add (charac r^a    |note charac ra does not work as
```

```
.                    |ra is not a dummy argument
.
.
```

The sequence upper case, 1, lower case is deleted during the macro definition, but causes the macro scan to search on each side for dummy arguments. In this case, a is found to be a dummy argument, and is treated accordingly. If the upper case 1 is not both preceded and followed by case shifts, only the 1 is deleted. Example:

```
define type x464pq
       lda (charac r1x464pq
       pno
       terminate
       type f     gives     lda (charac rf
                            pno
```

How may one cause a created symbol to define a variable? It will not do to write the dummy argument in upper case, for then the created symbol would be in upper case. Since upper case numerals are not legal symbol constituents, created symbols must not appear in upper case. The solution is to append a suitable upper case letter, say z, to the dummy argument.

Example:

```
define        macro |a
              sto aZ        |case shift makes end of
              tsx subr      |dummy argument a
              lda aZ
terminate
```

The variables would then be of the form 000a01Z, 0000a02Z, etc. which are perfectly legal and unique variables.

Created symbols have been introduced to solve the problem of address tags within macro definitions, but they may be used in other ways also. Some examples are given in Appendix 2.

Macro definitions may contain other macro definitions or macro calls. Arguments of the macro being called may be used in the macros it calls or defines with perfect generality. As an example, let us rewrite the write macro so that it inserts a suitable text printing subroutine into the object program at its first call, and then redefines itself so that later occurrences call the subroutine. This might be done as follows:

```
  define    write a
            define write c|d     |redefines write when called first time
            tsx wr
            d-.-2
            text |c|
  d,        terminate write
            write 6a9            |calls new definition
            tra zzxgwq
  wr,       llx 0                |text printing subr
            ixl
  lpkh,     ixl
            lax 1
            aux .-1
            pnt
            pnt
            prtUixl
            tix lpkh
            ixl
            trx 1
zzxgwq,     terminate
```

Notice that address tags in the text printing subroutine need not be created symbols, as the tags appear only at the first call of write. They must not, of course, conflict with tags used elsewhere in the program, and to insure this, created symbols may be used if desired. Notice that, in this example,

the pseudo-instruction terminate has been supplied with an argument: the name of the macro being defined. If terminate is followed by a space, it will expect to find this argument, which it will compare with the name of the macro being defined. Unless they agree, an error comment (mnd) will be made. This permits the programmer to be sure that his defines and terminates count out correctly. An additional aid in this respect is the fact that terminate is undefined outside a macro definition.

Arguments can, by judicious use of argument quotes (see example below), contain sub-arguments. A pseudo-instruction irp (indefinite repeat) permits the analysis of such an argument. The pseudo-instruction irp in the macro definition takes one argument, namely, the dummy argument corresponding to the argument to be analyzed. When the macro instruction is called, the characters following the argument of the irp until the next matching endirp will be inserted once into the program for each sub-argument in the argument being analyzed, and the sub-arguments will be substituted for the corresponding dummy argument. Example:

```
define      sum a,b,c
            lda a
            irp b
            add b
            endirp
            sto c
            terminate
            sum j, 6k,l,m9,N
gives:
            lda j
            add k
            add l
            add m
            sto N
```

It is quite permissible to have irp's within an irp, analyzing either the same or different arguments. The pseudo-instructions irp and endirp are defined only within a macro definition. If an irp analyzes a null string, the characters in the range of the irp will be inserted once, and null string will be inserted for the subargument.

The Garbage Collector

When MIDAS redefines a macro, the space in the macro instruction table used by the old definition will be recovered if necessary, by a garbage collector. It is important in a long program to insure that unused macro definitions are abandoned, that is, that their names are caused to refer to something else other than the original macro definitions. A suitable "something else" is the pseudo-instruction null, which does absolutely nothing. Thus if a macro called foo has been defined, it may be discarded after its last usage by saying:

```
equals foo, null
```

which will make the space used by foo recoverable. The garbage collector is called whenever the combined macro and symbol tables are exhausted. If no space can be recovered, an error comment is made (sce).

Repeat

The pseudo-instruction repeat expr, anything, where expr is a symbolic expression defined on Pass 1 and anything is any string of characters terminated by a carriage return, causes

anything to be inserted into the program a number of times, called the count, equals to the value of expr. The anything, called the range of the repeat, can be storage words, parameter assignments, macro calls (if not containing carriage return in an argument), other repeats, or anything else. If repeat is used in the range of a repeat, both repeats will end on the same carriage return. Repeat may be used in macros, and dummy arguments may appear either in the range or the count of the repeat, or both. If the count of a repeat is zero or negative, the range of the repeat is ignored.

Dimension

The pseudo-instruction dimension may be used to allocate space for arrays. The statement

dimension name1(size1), name2(size2),...,

causes space to be reserved in the variables storage for the array names specified. Each name is defined as the location of the first of the block of registers of the length specified. The array names must not have conflicting definitions elsewhere, and the array sizes must be defined at their occurence on Pass 1.

Conditional Assembly

It is often useful, particularly in macro instructions, to be able to test the value of an expression, and to condition part of the assembly on the result of this test. For this purpose the pseudo-instruction 1if and 0if are provided. Following the pseudo-instruction name there is a symbol called

a qualifier that determines the type of test; and then an argument that is tested according to the qualifier. The argument is ended by any of the word terminators tab, carriage return, comma, or slash. All these terminators except slash do what they would have done had the conditional not been present; but slash only marks the end of the conditional, which is treated as a single syllable whose value is one or zero. Examples:

```
repeat 0if vp x+1, macro arg1, arg2␤
a=1if vzxI600000⟶|
sto p+1if vp-s|X2␤
```

The value of 1if is one if the condition tested for is true, and zero otherwise; while the value of 0if is zero if the condition tested for is true, and one otherwise. There are at present two qualifiers with two corresponding tests:

vp: If the value of the expression following is positive or zero (either plus or minus), the test is true.

vz: If the value of the expression following is zero, the test is true.

The first example calls the macro if x≥-1. The second example defines a as one if the two high bits of x are both zero; otherwise a is defined as zero. The third example generates sto p if s is positive, and sto p+2 if s is negative. It could also be written as:

```
sto p+2X0if vps␤
```

Conditionals may be used in or out of macros, but may not contain other conditionals.

The Source and Object Programs

A source program for MIDAS consists of one or more flexo tapes, each with a title, a body, and a start pseudo-instruction. The title is the first string of characters and is terminated by a carriage return. Carriage return and stop codes preceding the title are ignored. The body is the storage words, macros, parameter assignments, etc. which make up the substance of the program. It may be void. The start pseudo-instruction denotes the end of the source program tape. It takes one argument, which specifies the first instruction to be executed in the object programs. Start must be preceded by a tab or carriage return, and followed (after the argument, if supplied) by a carriage return. READ THE LAST SENTENCE AGAIN. In spite of all warnings, the number of people who omit the carriage return after start is amazing. Therefore, take heed.

MIDAS will normally punch a binary object program during Pass 2 of an assembly. It will contain a title in readable characters, consisting of the visible characters in the title except those following (and including) an equals sign. Next will be punched an input routine, which is a loader that reads in the rest of the tape, and which is itself read in by the TX-0 read in mode. The binary output from the body of the source program is punched in blocks of up to 100 registers. The end of the binary tape is denoted by a start block, which is produced by the pseudo-instruction start. The start block may be of two types:

1. The add start block causes the input routine to stop, and pressing Restart transfers to the address specified. It is punched by start addr, where addr is a symbolic expression whose value specifies the starting address. MIDAS adds add to this and punches it on the tape.
2. The trn start block causes the input routine to transfer at once to the address specified. In this case the argument of start must have the value of add addr where addr is the address in question. MIDAS adds add (=200000) to this, giving trn (=400000) and punches it on the tape.

The format of the output is subject to considerable control by the programmer. The pseudo-instruction noinput suppresses punching the input routine. The pseudo-instruction readin suppresses the input routine and punches in readin mode until the next encountering of the pseudo-instruction noinput, which resumes punching in input routine format. The normal input routine occupies registers 17756-17777, but an input routine occupying registers 0-22 will be supplied by the pseudo-instruction frontloading, which, if used, must be the first thing on the English tape (after the title, of course). This pseudo-instruction causes the location counter to start at 30 instead of the usual 20.

For fabricating special tape formats or punching start blocks without stopping the assembly, the pseudo-instruction word is provided. Its argument or arguments, separated by commas and ended by a tab or carriage return, are punched directly on the object program tape, and do not affect the location counter.

The tape formats discussed so far are characterized by having a specific location in core assigned for each word in

the object program. MIDAS will also produce relocatable tapes, which, by means of a special loader, may be placed anywhere in memory. Before using this feature, described in the next section, the reader is advised to familiarize himself with Memorandum M-5001-34, which describes the relocatable loader and relocatable system.

Relocatable Programming

The pseudo-instruction relocatable directs MIDAS to assemble the object program in relocatable format and sets the location counter to relocatable 0. Address tags will be defined as relocatable symbols (relocation count +1) as long as the location is relocatable. Symbols defined by parameter assignment will have a relocation count equal to that of the expression to the right of the equal sign except that no symbol may have a relocation count exceeding one in magnitude. A location assignment puts the location to relocatable or absolute according to whether the relocation count of the location assignment is +1 or 0. Relocatable also suppresses punching an input routine, replacing it with a word trn 17000, which, when executed in the readin mode, transfers control to the entry of the BRS Loader. Storage words in relocatable mode may have relocation +1, -1, or 0; words in absolute mode may have relocation 0 only.

The pseudo-instruction exit is used to define symbols which are external to the program being assembled. The usage is

exit s1, s2, s3, ...,

which enters the symbols s1, s2, ... in the transfer vector and

defines them as the addresses they occupy there. Only the first three characters of these symbols are significant to the relocatable loader. These symbols must not be defined with a conflicting definition elsewhere or an error message (mdx) will be produced.

The pseudo-instruction entry is used to denote points in the program to which external programs may transfer control. The usage is:

entry s1, s2, s3, ...,

where the symbols s1, s2, ... must be defined as address tags elsewhere in the program. The symbols so declared are entered in the program card. Again, only the first three characters of such symbols are significant to the relocatable loader. For a program with both primary and secondary entries, the pseudo-instruction entry is used twice consecutively, first listing the primary and then the secondary entries. To the extent that the pseudo-instructions relocatable, entry, and exit are used, they must be used in that order, and no storage words may intervene between them. A program with no entry specified is a main program, and the pseudo-instruction exit will cause a program card to be punched with a name of +0, as required by the BRS loader. If neither entry nor exit is used, no program card will be provided. Since any program to be loaded by the BRS Loader must have a program card, it has been made possible to get a program card with a main program entry by using the pseudo-instruction entry with no arguments. The maximum number of

arguments of entry is 37; there is no limit on the number of arguments of exit.

In relocatable programs, the pseudo-instruction noinput will suppress punching the word trn 17000 at the head of the object program tape.

Format

MIDAS has few requirements on format. The user should be aware of the following:

1. Carriage returns and tabs are equivalent except in the title, in the range of a repeat, and after start. Extra tabs or carriage returns are ignored.
2. Backspace, the upper case numerals except 1, 6, and 9, and the unused characters of the flexo code, including blank tape with only the seventh hole punched, are illegal except in arguments of flexo code pseudo-instructions.
3. Stop codes and color shifts are ignored except in arguments of flexo code pseudo-instructions. Upper case 1, 6, and 9 are similarly ignored when not in macro calls or definitions.
4. Deletes are always ignored.

Many programmers have found that adherence to a fairly rigid format is of help in writing and correcting programs. The following suggestions have been found useful in this respect:

1. Place address tags at the left margin, and run instructions vertically down the page indented one tab stop from the left margin.

2. Surround address tags with color shifts. It looks nice.

3. Use only a single carriage return between instructions, except where there is a logical break in the flow of the program. Then put in an extra carriage return.

4. Forget that you ever learned to count higher than three; let MIDAS count for you. Do not say sto .+6; use an address tag. This will save grief when corrections are required.

5. Organize the program by pages, separating each page of flexo tape with a stop code and some tape feed. Let the page boundaries coincide with logical divisions of the program if possible. Fixing one bad page and splicing in a new one takes about as much time as reproducing two pages of program, so learn to splice tape.

6. Have the typescript handy when assembling or debugging a program, and note corrections in pencil thereon as soon as you find them.

Performing an Assembly without Magnetic Tape

First read in MIDAS. Turn on the on-line flexowriter and press Start Read. Set the TBR to trn 20 and TAC to 0. Load the first source tape into the reader and press Restart. MIDAS will read the tape in sections of about two pages each, and will stop shortly after reading start at the end of the tape. To process additional tapes after the first, press Test. Now begin Pass 2 by loading the first tape and pressing Restart. For additional tapes, press Test. At the end of Pass 2, press Restart again to secure a start block. Tapes should be processed in the same order on both passes.

The normal operation of MIDAS may be summarized by the following table:

Condition	AC	LR	MBR	Action on Restart	Action on Test
MIDAS or symbol punch read in	0	-0	-0	Begin Pass 1	Begin Pass 2
End of tape, Pass 1	0	0	0	Begin Pass 2	Continue Pass 1
End of tape, Pass 2	0	0	0	Punch start block	Continue Pass 2
After start block	0	-0	-0	Restore, begin Pass 1	Begin Pass 1
Error stop	-0	-0	-0	Continue, suppress punching	Continue Pass

The normal sequence of operations above can be modified by use of the TAC. Whenever Test is pressed, bit 0 of the TAC is examined. If it is zero, the normal sequence is followed; if it is 1, the next 6 bits of the TAC are examined. Note that bits 14-17 are examined regardless of the setting of bit 0. The other bits of the TAC control:

Bit 1 Pass 1 if 0, pass 2 if 1.

2 Begin pass if 0, continue pass if 1.

3 If 1, punch if pass 2; if 0, do not punch.

4 If 1, punch input routine if punching; if 0, no input. This takes precedence over the pseudo-instruction noinput.

5 If 1, punch title if punching; if 0, no title.

6 If 1, restore symbol table to initial symbols and pseudo-instructions.

14 (Only with mag tape MIDAS) If 1, input will be from mag tape; if 0 input method determined by TAC 15. (See page 29.)

15 If 1, input will be from the on-line flexowriter; if 0, from paper tape (see page 30).

16 If 1, printout on-line every character processed.

17 If 1, continue processing after an error stop. Equivalent to pressing Test.

Symbol Punch and Symbol Print

A record of symbol definitions may be printed out by reading in MIDAS SYMBOL PRINT.

A punched record of symbol and/or macro instruction definitions may be obtained by use of MIDAS SYMBOL PUNCH. When SYMBOL PUNCH is read in, it will feed some blank tape and listen for a title. Type a title on the typewriter. To obtain both symbol and macro-instruction definitions, terminate the title with a carriage return. For symbols only, terminate with a tab, and then type "s" followed by a carriage return. For macro definitions only, terminate the title with a tab, followed by "m" and a carriage return. The symbol punch so obtained may be used with DOCTOR for symbolic debugging, or read into MIDAS at a later time for assembling patches or the like. When a symbol punch is read into MIDAS, TAC 6 is examined. If off, the symbols from the symbol punch are merged with any existing symbol table. If on, the symbol table is restored to the initial vocabulary before merging the symbol punch.

A symbol punch may be read into MIDAS by placing the symbol table in the PETR and pressing "READ-IN." MIDAS will then read in the symbol table and halt when it is done. Then one may load his tapes which are to be assembled and process them. The symbol table may be read in at any time during an assembly. (However, reading it in <u>after</u> an error has occurred will <u>not</u> correct that error.)

The MIDAS Mag Tape System

The MIDAS Mag Tape System is an optional feature of the MIDAS programming system. MIDAS Mag Tape Patch is a program which is read into memory after MIDAS has been read in and which gives MIDAS the ability to use the magnetic tape unit both as an input and an output device. MIDAS Mag Tape Patch permits most English tapes to be assembled and ready to run in only a fraction of the time required when punched paper tape output is obtained. Moreover, the English tapes need be run through the PETR only once at assembly time, thus saving tape rewinding time and eliminating PETR errors on pass 2.

Most MIDAS English tapes can be assembled with the MIDAS Mag Tape Patch. However, the following additional restrictions apply:

1. Relocatable tapes cannot be assembled on mag tape.
2. The results of WORD pseudo-instructions do not appear in the mag tape output.
3. Standard mag tape format is always obtained on the mag tape output even though input mode format has been specified in the English program. (Mag tape format is described below.)
4. All tapes processed on pass 1 are reprocessed in the same order on pass 2 automatically. Although this is usually the case without the Mag Tape Patch, there is no longer any choice.
5. The binary program must contain the same number of words on pass 2 as it does on pass 1. No ordinary program will ever violate this restriction. This restriction does not apply to constants.

To use MIDAS Mag Tape Patch:

1. Read in MIDAS.
2. Read in MIDAS Mag Tape Patch.
3. Run pass 1 on all English tapes using the same procedure as with ordinary MIDAS. During pass 1 MIDAS will write out each gulp of the English tape onto mag tape. MIDAS leaves enough room between the blocks of English tape to write out the binary program later on pass 2.
4. Begin pass 2 by pressing RESTART when pass 1 is complete. There is no need to reload the English tapes in the PETR. MIDAS will rewind the mag tape and run pass 2 automatically. The title of each English tape will be printed out when it is encountered on the mag tape. The writing of the start block is automatic. If there are no errors in the English tape MIDAS will halt at the end of pass 2 with +0 in the accumulator and will begin another pass 1 when RESTART is pressed. If errors occur, pressing either RESTART or TEST will resume the assembly.
5. MIDAS checks the checksum of each record that it reads or writes. If an error is found, the program will rewrite (or reread) a block five times before giving up. If the error cannot be corrected, MIDAS will halt with -0 in the accumulator, live register, and MBR. Pressing RESTART will cause MIDAS to make five more tries. If MIDAS fails again, load a fresh reel of tape into the magnetic tape unit and begin pass 1 over again.

A punched paper copy of the binary program can be obtained by setting the TAC to 670000 and pressing TEST to start pass 2. A start block is automatically punched at the end of pass 2.

TAC 14 may be used to cause the input for pass 1 to be taken from the mag tape. This is useful if one is assembling several tapes together, and discovers an error on one of the tapes. When this tape has been corrected, put TAC 0 and TAC 14 up, TAC 1 and TAC 2 down, set any other appropriate TAC bits, and press TEST to perform pass 1 on the first tape with the input from mag tape. To take additional tapes from mag tape, leave the TAC set as it is except for putting TAC 2 up, and press TEST for each additional tape to be taken from the mag tape. When the corrected tape is to be read in, put TAC 14 down, and place the tape in the PETR and press TEST. Any additonal tapes must also be read in this manner; they cannot be taken from the mag tape. Then continue with pass 2 in the usual manner.

A "symbol punch" on mag tape is obtained by reading in MIDAS Symbol Punch at the end of pass 2. If bit 3 of the TAC is up at that time, a punched paper copy of the symbol punch will be obtained simultaneously as with paper tape MIDAS. The symbols on mag tape can be used by Doctor.

A symbol print can be obtained as with ordinary MIDAS. MIDAS mag tape format is as follows:

1. A record headed by a store-class instruction is a binary block and has the same format as a paper tape binary block (first address, complement of the last address, data, complement of the checksum).

2. A record headed by a transfer-class or add-class instruction is a start block as on paper tape.

3. A record headed by an operate-class instruction contains part of the English source tape which MIDAS read on pass 1.

The Mag Tape Input Routine program will read into memory a program having the MIDAS mag tape format. The Mag Tape Input Routine extends from register 17742 to register 17777. This is slightly longer than the standard input routine, and users whose programs use all of memory should take notice.

The MIDAS On-Line Input Feature

It is occasionally desirable to make simple corrections to an assembly without preparing additional input tapes off-line. This feature can be especially useful for defining symbols which were left undefined inadvertently, or to set parameters affecting the assembly. The MIDAS On-Line Input Feature enables the on-line flexo to communicate with MIDAS as if the input were coming from paper tape.

To use this feature set TAC 15 = 1, and press RESTART or TEST as appropriate. This may be done at any time during an assembly. Make sure PUNCH ON is depressed and begin typing.

MIDAS will buffer all input up to the word "start", and will then process the entire typed input. Should you make an error, set the TAC 0=0, press TEST, and begin anew. Make sure all inputs begin with a title and end with "start" followed by a carriage return. To return to paper tape operation, set TAC 15 = 0. This patch is compatible with the MIDAS Mag Tape System.

Error Stops

MIDAS will complain about various ambiguities and error conditions found in source programs. Some of these have already been mentioned. An error listing has the following format:

Column 1: A three letter code describing the type of error. A number following is the depth of macro calls.

2: The octal location in the object program. The symbol r means relocation.

3: The symbolic location in terms of the last address tag seen.

4: The last pseudo- or macro-instruction name seen.

5: The offending symbol, if a symbol was in error.

MIDAS will ignore most errors (with exceptions noted below) and will continue the assembly if Restart or Test (with TAC 0=0) is pressed; the two are equivalent except Restart will discontinue punching on Pass 2 if it was in progress. Turning up TAC 17 is equivalent to pressing Test after an error stop. This bit is independent of the rest of the TAC.

The error conditions are:

us- : In general, undefined symbol. Undefined symbols are evaluated as 0. The third letter tells where it was found.

w: In a storage word or argument of pseudo-instruction word.

m: In a storage word generated by a macro call.

d: In the size of a dimension array.

p: In a parameter assignment.

c: In a constant.

s: In the argument of start.

e: In an argument of entry.

r: In the count of a repeat.

t: In an address tag of more than one syllable. This will frequently be the result of an undefined macro instruction.

i: In an argument of 0if or 1if.

ich Illegal character. The bad character is ignored.

ilf Illegal format. Some character or characters were used in an improper manner. Characters are ignored to next tab or carriage return.

ile Illegal entry. Argument of entry is improper and will be ignored.

ilx Illegal exit. Argument of exit is improper and will be ignored.

ir- : Illegal relocation. The relocation is taken as 0. The third letter identifies where it was found, and will be the same as listed under undefined symbols (above).

mnd: Macro name disagrees. The argument of terminate disagrees with the name of the macro being defined. First name is used.

mdt: Multiply defined tag. Original definition retained.

mdx: Multiply defined exit. An argument of exit is previously defined with a conflicting value. Orginal definition retained.

mdv: Multiply defined variable. A symbol containing an upper case letter is previously defined as other than a variable. Original definition retained.

mdd: Multiply defined dimension. An array name in a dimension statement has a conflicting definition. Original definition retained.

ipa: Improper parameter assignment. The expression to the left of an equal sign is improper. The assignment is ignored.

sce: Storage capacity exceeded. Assembly cannot continue.

tmc: Too many constants: the pseudo-instruction constants used more than 10. times in one program.

tmp: Too many parameters: the storage reserved for macro instruction arguments has been exceeded.

tme: Too many entries. Maximum number of arguments of an entry pseudo-instruction is 37 octal.

tmv: Too many variables. The pseudo-instruction variables has been used more than 8 times in one program. Assembly cannot continue.

cld: Constants location disagrees. The pseudo-instruction constants has appeared on Pass 2 in a different location from that found on Pass 1, meaning all the constants syllables have been assigned the wrong value. Assembly cannot continue.

cad: Constants area deficient. Insufficient space was saved for constants when the pseudo-instruction constants was encountered on Pass 1, and any following program will overlap the constants area used, unless given a specific location assignment by means of |. The condition is ignored.
Two possible recoveries are finding where the constants area ends (by means of the SYMBOL PRINT), making a location assignment on the English tape after the constants pseudo-instruction, and reassembling the tape, or making a patch with DOCTOR to replace the constants that have been clobbered.

vld: Variables location disagrees. The pseudo-instruction variables has appeared on Pass 2 in a different location from that found on Pass 1. The condition is ignored.

iae: Internal assembler error. MIDAS has found that it has made a mistake in assembling the program. Deliver the error message and a copy and listing of the source program to a member of the TX-0 staff so that the trouble may be found. Assembly cannot continue. The octal location given is the location in MIDAS where the error was found.

Troubleshooting

The checking features built into MIDAS will detect simple errors like forgotten tags very simply. Attempting to debug complex macro definitions from error messages and binary output is a much more difficult proposition. Special aids have been provided to simplify this.

1. The pseudo-instructions print and printx take an argument exactly like text, which MIDAS will print out online during the assembly process. Printx prints just the argument and a following carriage return, while print precedes this with the first three columns of an error listing, with the "error" code pnt. The argument of print or printx may contain dummy symbols if used in a macro definition.

2. Bit 16 of TAC when on, causes MIDAS to print out online every character it processes, including all macro expansions. This permits the programmer to let MIDAS do the bookkeeping when testing a complicated macro.

Appendix 1. MIDAS Initial Vocabulary

Part 1. Symbols

```
add=200000
ado=060000
adx=220000
alc=640260
all=640230
alo=640220
alr=640200
alx=640031
amz=640050
ana=740027
anl=640207
ano=740207
arx=640601
aux=260000
axc=640061
axo=640021
axr=640001
bsr=604000
cal=700200
cax=700001
cla=700000
clc=700040
cll=631000
clr=632000
com=600040
cpf=607000
cpy=620000
cry=600012
cyl=640030
cyr=600600
dis=622000
dso=662020
ex0=610000
ex1=611000
ex2=612000
ex3=613000
ex4=614000
ex5=615000
ex6=616000
ex7=617000
hlt=630000
iad=640232
ial=740222
ixl=600303
lac=700022
lad=600032
lal=700012
lar=700622
lax=360000
laz=700072
lcc=700062
lcd=600072
lda=340000
ldx=240000
llr=300000
llx=320000
lpd=600022
lro=600200
lxr=600003
nop=600000
opr=600000
ora=740025
orl=640205
oro=740205
p6b=766020
p6h=626600
p6o=666020
p6s=726000
p7h=627600
p7o=667020
pen=603000
pnc=664060
pno=664020
pnt=624600
prt=624000
rlc=721000
rlr=721600
r3c=723000
rax=640203
rds=604004
rew=604010
rpf=706020
rtb=604004
rtd=604024
rxa=600322
shr=600400
slr=100000
slx=120000
spf=647000
sto=0
stx=020000
stz=140000
sxa=040000
tac=701000
tbr=702020
tix=460000
tlv=540000
tpl=560000
tra=500000
trn=400000
trx=520000
tsx=440000
typ=625000
tze=420000
wrs=604014
wtb=604014
wtd=604034
xac=700120
xad=600130
xal=700110
xcc=700160
xcd=600170
xlr=600300
xro=600001
```

APPENDIX I--MIDAS INITIAL VOCABULARY

Part 2--Pseudo-Instructions

character	Inserts numerical value of a flexo character. (8)
constants	Denotes location of stored constants words. (7)
decimal	Interpret integers as decimal numbers. (3)
define	Define macro-instructions. (10)
dimension	Allocates space for arrays. (17)
endirp	Ends indefinite repeat. (15)
entry	In relocatable programs, puts symbol definitions into program card for use by BRS loader. (22)
equals	Defines symbol as operationally equivalent to another symbol. (5)
exit	In relocatable programs, names subroutines to be called by entering names in transfer vector. (21)
flexo	Inserts numerical value for three flexo characters. (8)
frontloading	Calls for front input routine. (20)
irp	Indefinite repeat. Analyses macro-instruction argument as series of subarguments. (15)
noinput	Suppresses input routine, leaves "readin" status. (20)
null	No-operation, ignored. (16)
octal	Interpret integers as octal numbers. (3)
opsyn	Defines symbol; same as equals but effective on Pass 1 only. (5)
print	Generates symbolic location printout and prints comment during assembly. (34)
printx	Prints comment during assembly. (34)
readin	Punch in readin mode format. (20)
relocatable	Punch in relocatable format. (21)

APPENDIX I--Part 2 Cont'd.

repeat	Repeats character string. (16)
start	Denotes end of program and specifies (in absolute program) starting address. (19)
terminate	Ends macro definition. (10,15)
text	Inserts words of flexo characters. (8)
variables	Reserves space for arrays and variables. (5)
word	Punches word on object program tape. (20)
0if	Has value 0 if condition following is true, 1 otherwise. (17)
1if	Has value 1 if condition following is true, 0 otherwise. (17)

APPENDIX II

SOME MACRO-INSTRUCTION EXAMPLES

Following are some examples illustrating some more complex uses of macro-instructions. All of these examples use so-called "information carrying macros." Basically, an information carrying macro is a name assigned to a character string which has provision for using or modifying the string. Three different methods are used for retrieving the information in the following examples.

The first two examples illustrate a method of locating coding at a remote place in the program. It is sometimes convenient, in the middle of a program, to specify flexo text, subroutines, or other material to be inserted at an out-of-the way place. The macro name remote, followed by arbitrary material as an argument, saves up such material for all users of remote until the macro-instruction here is used, which unloads all the stored information into the program at that point.

In the first example, listname is the information carrying macro. Each call of remote calls in cons to concatenate the new information onto the end of the old. The key to understanding the example is in the definition and use of listname. In order to feed the information in listname into some macro which can make use of it, listname must be called (expanded) and the characters therein fed to the macro to make use of them. This is done by feeding the name of the macro to use the information to listname as its argument. The expansion of listname generates the name of the user, followed by two arguments: the name listname itself, followed by the information characters in listname.

Thus the user macro can be one which deals with several different information carriers, each of which carries its own label. The point is that in order to generate a function of the information in an i.c.m., first take i.c.m. name of the function name. The i.c.m. flips the function name in front of the information as it expands.

Exercise: Generate the expansion of the following code:

```
remote alfa
remote `add T
       sto T'
here
```

The second example has remote as the i.c.m. The definition of remote is such that remote effectively redefines itself, adding on to its definition anything fed it as an argument. The here macro redefines listname so that when remote next calls it, it unloads itself into the program instead of into a new definition of remote. The definitions as written here are not self-resetting: the appearance of here does not leave either remote or listname in condition to be used again.

Exercise: Define a macro setup which establishes the correct initial definitions of remote and listname when it is called. Insert calls of setup in appropriate places so that the definitions of remote and listname are properly initialized, and are reset by use of here.

The purpose of the third example is to allow indiscriminate use of the pseudo-instructions octal and decimal in macro definitions without disturbing the current radix outside of macro calls. To this end, the system definitions of octal and

decimal are saved in the name roctal (real octal) and rdecml (real decimal). Then octal and decimal are defined as macros which, in addition to setting the current radix, also append to radix to a list of radices called list. To restore the previous radix, the macro oldradix peels the top entry off the list and discards it, then sets the current radix to the top of the entry of the remaining list. The list, after use of decimal and octal would look in part like this:

```
define append newrdx
list newrdx, 6roctal9, 6rdecml, 6roctal, 6error999
terminate
```

The method used for manipulating the list is similar to that of example 2. Note how the third argument of list is added to and deleted from.

Exercises: Determine the definition of oldradix corresponding to the above definition of append. Expand decimal and determine its effect on the list.

The last example illustrates the use if irp, 0if and 1if. The macro deciprt prints out on-line at assembly time the numerical value of its argument in English words. Zero suppression, sign, and numbers ending in "teen" are all handled correctly. The i.c.m. info contains the text to be printed out, and is handled similarly to listname in the first example. The sequence info redefine appears so often in the original that the macro in has been defined as a shorthand for it. The conversion to decimal is handled by the usual method of depletion of powers of ten. Zero suppression is handled by the indicator sup.

```
|remote macros-method 1, S. R. Russell

define remote a
          listname 6cons 6a 9,9
termin

define listname user
          user listname,
terminate

define cons i2, name, i1|user
          define name user
          user name, 6i1
i2 9
terminate name
terminate cons

define here
          listname 2ndarg
          define listname user
          user listname,
terminate listname
terminate here

define 2ndarg a,b
          b
termin

start


|remote macros-method 2, A. Kotok

define remote a
          listname a
termin

define listname i1|i2
          define remote i2
          listname 6i1
i2 9
terminate remote
terminate listname

define here
          define listname info
info
terminate listname

remote
terminate here

start
```

```
|octal-decimal pushdown, S. D. Piner

opsyn roctal, octal
opsyn rdecml, decimal

define octal
          append roctal
termin

define decimal
          append rdecml
termin

define error
          print|Too many oldradix pullups.|
          list roctal, error
termin

define list radix, prevrdx, rdxlist
          define append newrdx
          list newrdx, 6radix9, 6prevrdx, 6rdxlist99
newrdx
terminate append
define oldradix
          list prevrdx,rdxlist
          prevrdx
terminate oldradix
terminate list

list roctal, error

start
```

```
|decimal print macros, D. A. Gross

define deciprt number
          z=number
          repeat 0if vp z,in minus      deci2 -z
          repeat 1if vp z| -1if vz z,deci2 z
          repeat 1if vz z| I1if vz zI1,in zero
          repeat 1if vz z| I1if vz zU1,in minus zero
          info write
          redefine
terminate

define deci2 a
          x=a
          sup=0
          deplete 100000.
          teen=0
          integer
          place hundred
          deplete 10000.
          intergy
          deplete 1000.
          integer
          place thousand
          deplete 100.
          teen=0
          sup=0
          integer
          place hundred
          deplete 10.
          intergy
          deplete 1
          integer
terminate

define redefine y
          define info user, data
          user y data
          terminate info
terminate redefine

redefine

define in a
          info redefine,a
terminate

define arg a,b
          sup=1
          repeat teen,in a
          repeat 1-teen,in b
terminate
```

```
define deplete a
          y=0
          repeat 9,repeat 1if vp x-a, x=x-a       y=y+1
terminate

define integer
int1 ‘‘arg eleven,one’, ‘arg twelve,two’,  arg thirteen,three’
‘arg fourteen,four’, ‘arg fifteen,five’, ‘arg sixteen,six’
‘arg seventeen,seven’, ‘arg eighteen,eight’, ‘arg nineteen,nine’’

repeat 1if vz y, repeat teen, in ten
terminate

define intergy
int1 ‘teen=1,in twenty,in thirty,in forty,in fifty,in sixty
in seventy,in eighty,in ninety’

repeat 0if vz y, sup=1
termin

define int1 k
          j=1
          irp k
          repeat 1if vz y-j,k
          j=j+1
          endirp
terminate

define write b
          printx |b|
terminate

start
```